# Image To Symbols

A **technical report** describing the algorithmic structure, mathematical properties, and information-theoretic implications of the `ImageToSymbols` class.
The goal is to clarify how it converts a real-valued vector (e.g., an image flattened into a 1-D float array) into *symbols* where **each bit of every symbol carries equal informational weight**, and how the inverse procedure recovers the image.

---

## 1. Overview

The `ImageToSymbols` class defines a **bidirectional transform**:

- **Forward transform (`imageToSymbols`)**
  Maps a real-valued input vector `image[vecLen]` to an array of unsigned integer `symbols[vecLen]`, each symbol holding `bitsPerSymbol` bits.

- **Inverse transform (`symbolsToImage`)**
  Iteratively reconstructs a floating-point vector that closely matches the original `image` from the integer `symbols`.

Key idea:

> **Use the Fast Walsh–Hadamard Transform (FWHT)** plus deterministic sign randomization to "flatten" the significance of bits.
> After transformation, *each bit position of every output symbol is derived from a different orthogonal ±1 pattern*, so no bit carries more inherent weight than any other.

---

## 2. Core Components

### 2.1 Pseudo-random Sign Mixing

- Each call to WHT is preceded or followed by multiplying selected elements by ±1 depending on the evolving 32-bit integer `r`.

`r` is advanced by a **linear congruential generator (LCG)**:

```
r = r * rmult + radd
```

- with carefully chosen `rmult`, `radd`, and the modular inverse `rmultInv`.
- The sign flips scramble spatial correlations while remaining perfectly reversible because:
    - forward: `r = r * rmult + radd`
    - inverse: `r = (r - radd) * rmultInv`
    - same seed → identical sequence.

## 2.2 Fast Walsh–Hadamard Transform

- `WHT.whtN()` implements the O(N log N) Walsh–Hadamard transform:
    - Orthogonal, real, and **self-inverse**.
    - Coefficients are ±1 only, making sign manipulations easy.
- Each transform redistributes energy so that subsequent thresholding of signs extracts independent ±1 “features”.

## 2.3 Bit-plane Extraction

Forward pass repeats `bitsPerSymbol` times:

1. Apply LCG-driven sign randomization.
2. Apply FWHT.
3. Threshold each coefficient at 0:
    - negative ⇒ set bit i of `symbols[j]`.

The *j-th symbol* accumulates bits from successive rounds, but the **bit index `i` is not tied to a power-of-two weight**—it is just an independent Bernoulli feature.

---

# 3. Information-Theoretic Significance

## 3.1 Equal Informational Weight

In normal binary representation, the most significant bit (MSB) carries exponentially more numeric weight.
Here, *each bit of each symbol* is a **statistically independent sign test** of an orthogonal linear projection.

- All bits contribute identically to reconstructing the signal.
- There is no fixed “value” per bit; each bit is a dimension in a ±1 hypercube.

## 3.2 Relation to Error-Correcting and Random Projections

- The method resembles **sign random projections** used in locality-sensitive hashing.
- The multiple WHT layers are akin to a *Hadamard code* or a **multi-round spreading sequence**.
- Each bit can be viewed as one row of a larger orthogonal matrix chosen via the deterministic RNG.

---

# 4. Inverse Algorithm

`samplesToImage` performs a kind of **projected gradient descent** in the Walsh domain:

1. **Random start**:
   `image[i] = rng.nextFloatSym() * 1.73205` (approx. √3 to match variance of ±1 sequences).
2. For each iteration:
    - For each bit plane `j`:
        - Apply the same sign randomization & WHT.

        - Flip `image[k]` signs if the current sign disagrees with the stored bit.
    - Run WHT again and *reverse* the sign randomization using the modular inverse LCG.
3. Final de-scrambling and one more WHT yields the reconstructed vector.

Because WHT is orthogonal and sign operations are involutions, the process converges to a signal whose signs match the measured bits.

---

# 5. Notable Properties

| Property | Consequence |
|---|---|
| **Perfect invertibility (no quantization)** | Given enough iterations, reconstruction is exact in floating-point arithmetic. |
| **Bit significance equalization** | Any single-bit error distorts output with the same expected energy as any other. |
| **Deterministic randomness** | Same seed guarantees one-to-one mapping for encryption-like reproducibility. |
| **Orthogonal spreading** | Good resilience to local noise, similar to spread-spectrum or Hadamard coding. |

---

# 6. Applications

- **Perceptual hashing / robust watermarking** – equal-importance bits are attractive for error detection.
- **Compression or coding research** – exploring representations where bits are exchangeable.
- **Privacy or encryption primitives** – reversible, pseudo-random scrambling of image data.
- **Bit prediction machine learning algorithms -** where the equal information in each symbol bit can make prediction easier.

---

# 7. Implementation Notes

- `vecLen` should be a power of two for FWHT.
- Numerical stability: since only signs are used, it tolerates moderate floating-point noise.
- `seed` uses `Mix13.mix(System.nanoTime())`; reproducibility requires saving the chosen seed.

---

## Summary

This code builds a *symbolic representation* of an image by repeatedly applying **Hadamard-domain sign tests under pseudo-random phase flips**.
Unlike standard binary encoding, **every output bit represents an independent ±1 feature of the original signal**, granting **equal informational meaning per bit** and allowing **lossless reconstruction** through the carefully inverted sequence of transforms and sign operations.

Java Source Code:

```
public class ImageToSymbols {
   final int vecLen;
   final int bitsPerSymbol;
   final int seed;
   final float[] work;
   final int rmult=0x93d765dd;
   final int rmultInv=0x7aae1a75;
   final int radd=0x9E3779B9;
   Rand rng=new Rand();

   public ImageToSymbols(int vecLen,int bitsPerSymbol) {
      this.vecLen=vecLen;
      this.bitsPerSymbol=bitsPerSymbol;
      work=new float[vecLen];
      seed=(int)Mix13.mix(System.nanoTime());
   }

   public void imageToSymbols(int[] symbols,float[] image) {
      System.arraycopy(image,0,work,0,vecLen);
      java.util.Arrays.fill(symbols,0);
      for(int i=0,r=~seed; i<vecLen; i++) {
         r=r*rmult+radd;
         if(r<0) work[i]=-work[i];
      }
      WHT.whtN(work);
      for(int i=0,r=seed; i<bitsPerSymbol; i++) {
         for(int j=0; j<vecLen; j++) {
```

```
            r=r*rmult+radd;
            if(r<0) work[j]=-work[j];
         }
         WHT.whtN(work);
         int bit=1<<i;
         for(int j=0; j<vecLen; j++) {
            if(work[j]<0f) symbols[j] |=bit;
         }
      }
   }

   public void symbolsToImage(float[] image,int[] symbols,int iterations) {
      for(int i=0; i<vecLen; i++) {
         image[i]=rng.nextFloatSym()*1.73205f;
      }
      for(int i=0,r=seed; i<iterations; i++) {
         for(int j=0; j<bitsPerSymbol; j++) {
            for(int k=0; k<vecLen; k++) {
               r=r*rmult+radd;
               if(r<0) image[k]=-image[k];
            }
            WHT.whtN(image);
            int bit=1<<j;
            for(int k=0; k<vecLen; k++) {
               if((image[k]<0f)^((symbols[k]&bit)!=0)) {
                  image[k]=-image[k];
               }
            }
         }
         for(int j=0; j<bitsPerSymbol; j++) {
            WHT.whtN(image);
            for(int k=vecLen-1; k>=0; k--) {
               if(r<0) image[k]=-image[k];
               r=(r-radd)*rmultInv;
            }
         }
      }
      WHT.whtN(image);
      for(int i=0,r=~seed; i<vecLen; i++) {
         r=r*rmult+radd;
         if(r<0) image[i]=-image[i];
      }
   }
}

public final class Mix13 {
   public static long PHI = 0x9E3779B97F4A7C15L;
   public static int PHI_INT=0x9E3779B9;
   public static long mix(long z) {
      z = (z ^ (z >>> 30)) * 0xBF58476D1CE4E5B9L;
      z = (z ^ (z >>> 27)) * 0x94D049BB133111EBL;
      return z ^ (z >>> 31);
   }
   long murmur64(long h) {
      h ^= h >>> 33;
      h *= 0xff51afd7ed558ccdL;
      h ^= h >>> 33;
      h *= 0xc4ceb9fe1a85ec53L;
      h ^= h >>> 33;
      return h;
   }
```

```
    public static long hash(long h,long x) {
        h+=x;
        return h*0xf1357aea2e62a9c5L;
    }
    public static int hashInt(int h,int x){
                            h+=x;
                            return h*0x93d765dd;
                }
    public static int mixInt(int h){
                    h ^= h >>> 17;
        h *= 0xff51afd7;
        h ^= h >>> 17;
        h *= 0xc4ceb9fd;
        h ^= h >>> 17;
        return h;
                }
}

public class WHT {
    public static void whtN(float[] x) {
        final int n=x.length;
        int hs = 1;
        while (hs < n) {
            int i = 0;
            while (i < n) {
                int j = i + hs;
                while (i < j) {
                    float a = x[i];
                    float b = x[i + hs];
                    x[i] = a + b;
                    x[i + hs] = a - b;
                    i += 1;
                }
                i += hs;
            }
            hs += hs;
        }
        float scale=1f/(float)Math.sqrt(n);
        for(int i=0; i<n; i++) {
            x[i]*=scale;
        }
    }
}
```